Problem Solving

- state space
  - graph whose nodes correspond to problem situations
- problem is reduced to finding a path in this graph
- blocks problem
  
  A
  B

Problem Solving

- can only move one block at a time
- block can be grasped only when top is clear
- block put on table or other block
- must find sequence of moves to satisfy the requirements
- problem is an exploration of alternatives
  - initially, one alternative - move C to table
  - then three alternatives - A on table, A on C or C on A

Problem Solving

- problem shows two concepts
  - problem situations
  - legal moves transforming problem situations into other situations
- these together form a directed graph - state space
  - nodes of a state space graph correspond to problem situation
  - arcs correspond to legal transitions between spaces
- finding a solution is equiv to finding a path from the initial state to the final (goal state)

Problem Solving

- can represent the 8 puzzle in a similar fashion
  - 8 tiles in a 9 square grid
  - given this starting point, draw a state space graph

Problem Solving

- useful for many practical problems
  - towers of hanoi
  - travelling salesman
    - practical optimisation problem
    - find shortest route from starting city and visit all cities
    - cannot visit city more than once (Except the start city which is the end point as well)
  - farmer, goose, fox and grain
    - boat only holds the farmer and one other
    - protect the grain from the goose from the fox
Problem Solving

- state space - defines the rules of the game
- nodes - situations in the state space
- arcs - legal moves between nodes
- can attach cost to moves
  - some blocks are harder to move than others
  - distance between towns
- with costs
  - problem becomes one interested in minimum cost

Problem Solving

- state space in Prolog
- \( s(X,Y) \)
  - true if there is a legal move in the state space from a node \( X \) to a node \( Y \)
  - \( Y \) is a successor of \( X \)
  - with costs, add a third argument
    - \( s(X,Y,Cost) \)
  - relation can be represented in the program explicitly by a set of facts
    - in most real problems, this is not feasible

Problem Solving

- \( s(X,Y) \)
  - would be defined implicitly by stating the rules for computing successor nodes
- how are nodes represented
  - compact enabling efficient execution of operations
    - evaluation of successor relation and associated costs
- back to the blocks
  - can represent the problem situation (node) as a list of stacks
  - each stack is a list of blocks

Problem Solving

- initial situation
  - \([c,a,b],[],[]\)
  - the list of blocks is ordered so that top block is at start of list
- goal is any arrangement with the ordered stack of all the blocks - three possibilities
  - \([a,b,c],[[],[]]\)
  - \([[],[a,b,c],[[]]\]
  - \([[],[],[a,b,c]]\)

Problem Solving

- successor relation
  - Situation 2 is a success or Situation 1 if there are two stacks (Stack 1 and Stack 2) in Situation 1 and the top block of Stack 1 can be moved to Stack 2
- in prolog
  \[
  s(Stacks, [Stack1, [Top|Stack2] | OtherStacks]) :-
  del([Top|Stack1], Stacks, Stacks1),
  del(Stack2, Stacks1, OtherStacks).
  \]

Problem Solving

- goal condition
  - goal(Situation):- member([a,b,c],Situation)
- program search algorithms as a relation
  - solve(Start, Solution)
    - Start is the start node in state space
    - Solution is a path between Start and any goal node
      - solve([c,a,b], [], []). Solution
Depth First Search

- to find a solution path, Sol from a given node N to some goal node
  - if N is a goal node, then Sol = [N]
  - if there is a successor node N1, of N, such that there is a path Sol1 from N1 to the goal node, then Sol = [N|Sol1]
- translates to Prolog

\[
\text{solve}(N, [N]) :- \text{goal}(N).
\]
\[
\text{solve}(N, [N|Sol1]) :- \text{successor}(N, N1), \text{solve}(N1, Sol1).
\]

Depth First Search

- order in which alternatives are explored
- when presented with a choice of nodes, it chooses the one furthest from the start node
- searches down the tree first
- works well in many situations
- is simple to program due to the recursive nature
  - 8 queens is an example of DFS
- alternative solutions found through backtracking

Depth First Search

- many ways in which DFS can run into trouble
  - adding an extra arc from h to d will result in infinite loop for search
  - solution is to add a Path variable to detect where we have been
    - any node in the path from the start node to the current node should not be considered again
  - Path variable serves two purposes
    - prevent infinite loops, or second visits
    - build a solution path

Depth First Search

- other problem
  - many state spaces are infinite
  - DFS may miss a goal node and proceed along an infinite branch
    - then explores here without ever getting to the goal
- to overcome this, can add a maximum depth of search
  - limits the depth
  - on every recursion, decrease the maxdepth, but never let it get negative (ie stop going down when zero)

Depth First Search

- iterative deepening
  - varying the depth of the search to avoid the problems of setting limit too low or too high
    - can start with low search, increase to larger values until solution found
- write a depth first strategy
Breadth First Search

- chooses to visit nodes closest to start node first
- search across instead of down
- not as easy to program
  - have to maintain set of alternative candidates
  - will continually grow
  - also need to maintain the Path list

Searching to find path \([a, b, e, j]\) will find \([a, c, f]\) first

Breadth First Search

- outline for search
  - if head of first path is goal node, then this path is a solution
  - otherwise, remove first path from the candidate set and generate the set of all possible one-step extensions to this path
    - add this extension at the end of the candidate set and execute BFS on this updated set
- initial candidate \([a]\)
- generate extensions \([b, a], [c, a]\)

Breadth First Search

- remove path \([b, a]\) and generate extensions
  - \([d, b, a], [e, b, a]\)
- add these back
  - \([c, a], [d, b, a], [e, b, a]\)
- remove \([c, a]\) and generate extensions
  - \([d, b, a], [e, b, a], [f, c, a], [g, c, a]\)
- after removing and extending \([d, b, a]\) and \([e, b, a]\) get
  - \([f, c, a], [g, c, a], [h, d, b, a], [i, e, b, a], [j, e, b, a]\)

Breadth First Search

- path \([f, c, a]\) has goal node \(f\)
  - path returned as solution
- write a breadth first search
- consider the differences between DFS and BFS
  - which will give the shortest solution
  - how does DFS iterative deepening affect your answer